Lecture 01. Compilers Overview

Machine Lang(machine instructions; patterns of 0's and 1's) -> Assembly Lang -> Higher-level lang(C,C++,Java...) (easier to develop; 인간 친화적으로 발달해 옴)

Higher-level lang을 computer에서 실행시키려면, **Language translation**이라는 추가적 과정

1. Language translation -> source code(C, C++, java, python)를 **semantically-equivalent**(의미가 같은) **target code**(ex. Assembly, machine lang)으로 translate

2. Error detection -> translation process 동안, source program 내의 error를 detect & report

Source code => language processors => target code

(Detect errors) => Error messages (+report)

	document translation ≒ Compilation	live translation ≒ Interpretation
What to translate	xxx.c An entire source program	One statement of a source program xxx.py
When to translate	Once before the program runs 실행전 translate	Every time when the statement is executed
Translation result	xxx.exe A target program ← entire (equivalent to the source program)	Target code (equivalent to the statement)
Examples	C, C++	Javascript , Python

	Compilation	Interpretation
Runtime performance (execution time)	O	need compilation during run-time
Portability / flexibility	ex. Intel vs ARM vs Intel (window) (window) (Linux)	O
Debugging / development	Translate entire source code whether there's error or not ex. good for released , optimized	Can translate only modified part ⇒ reduce debugging & development time ex. good for prototype

<Hybrid Compilers> combine compilation and interpretation (Java, Python ex. pyc)

Make intermediate program (ex. bytecode) => more computer-friendly but not machine level -> reduce overhead and increase run-time performance and keep portability

<Common Language-processing systems>

Src prgm(test.c) ==(Preprocessor)==> Modified src prgm(optimized) ==(Compiler)==> Target assembly prgm(test.s) ==(Assembler)==> reloctable machine code(test.o) ==(Linker)==> Absolute machine code (executable binary file; test.out)

<Requirement for good compilers>

1. Correctness (mandatory) **(MAJOR)** 2. Performance improvement (optional) 3. Reasonable compilation time (optional)

Modern compilers preserve the outlines of the FORTRAN 1 compiler

Using Symbol table (used by all phrases of compilers)

Lexical analyzer (scanner) -> Syntax analyzer (parser) -> Semantic analyzer (Analysis part)

-> Intermediate code generator -> Code optimizer -> Code generator (Synthesis part)

<Lexical analyzer (scanner)> -> divide the stream of characters into meaningful sequences and produce set of tokens (A=B+C => 'A' '=' 'B' '+' 'C')

<Syntax analyzer (parser)> -> tree-like intermediate representation (syntax tree) that depicts the grammatical structure of the token stream

Lecture 02. Lexical Analysis (specification of tokens)

- **Token**: syntactic category (ex. Identifier, number, operator, ...)

(token name, token value) pair로 structrured (token value는 optional)

(Keyword: {IF, ELSE, FLOAT, CHAR 등}, Operators: {ADD, COMPARISON 등}, Identifiers: {ID}, Constants: {NUMBER, INTEGER, REAL, LITERAL 등}, Punctuation symbols{LPAREN, COMMA 등}, Whitespace: {non-empty sequence of blanks, newlines, tabs 등 ex. 주석, 빈 칸})

- **Lexemes**: sequence of characters that matches the pattern for a token

Ex. i -> ID, if -> IF, 3.14->NUMBER, (-> LPAREN, "Hello" -> LITERAL ...

- **Lexical Analyzer does?**: 1. Partitioning input strings into substring (lexemes) 2. Identifying the token of each lexeme

Input	A	=	B	+	C
Token name	ID	ASSIGN	ID	ADD	ID
Token value	A or pointer to symbol-table entry for A		B		C
Output	<ID, A>	<ASSIGN>	<ID, B>	<ADD>	<ID, C>

- How to specify the patterns for tokens? -> Regular languages

- How to recognize the tokens from input streams? -> Finite Automata

Regular Languages(⊂ Context-free lang ⊂ Context-sensitive lang ⊂ Recursively enumerable lang) -> Simple but powerful

- alphabet Σ -> finite set of symbol (ex. Letter = $\Sigma^L = \{A, \ldots, Z, a, \ldots, z\}$, Digit = $\Sigma^D = \{0, \ldots, 9\}$)

- string s -> s over alphabet is a finite set of symbols drawn from the alphabet

(string: $\Sigma = \{0\} \to s = 0, 00, 000, or, \ldots$ $\Sigma = \{a, b\} \to s = a, b, aa, ab, ba, bb, aaa, or \ldots$)

- language L -> any set of strings over some fixed alphabet Σ

(language: $\Sigma = \{a, b\} \to L_1 = \{a, ab, ba, aba\}$ $L_2 = \{a, b, aa, ab, ba, bb, aaa, \ldots\}$)(L1 finite, L2 inf)

Operation s, |s| (length) , s_1s_2(concatenation), ϵ (empty string), s^i(s의 expo;concat i-times)

Operation L, $L_1 \cup L_2$ (Union), L_1L_2 (Concatenation), L^i (Concat of L i-times), L^* (kleene closure; 0 or more), L^+ (Positive closure; one or more)

Regular expression r -> regular language L(r)

$\epsilon \to L(\epsilon) = \{\epsilon\}$ $a \to L(a) = \{a\}, a\ in\ \Sigma$, $r_1|r_2 \to L(r_1) \cup L(r_2)$ $r_1r_2 \to L(r_1r_2) = L(r_1)L(r_2)$

$r^* \to L(r^*) = \cup_{\{i \geq 0\}} L(r^i)$ ex. $a^+ = aa^*$ but $(^n)^n\ \Sigma = \{(,)\}$ -> RE로 불가능

Rules for RE Precedence: $(.^*, .^+)$ > concat > | ; Equiv: same exp -> same lang

| : Commutative, Associative Concat: Associative, Concat distribution over |

ϵ: identity for concat ($r_1\epsilon = \epsilon r_1 = r_1$), guaranteed in * ($r^* = (r|\epsilon)^*$) $a^{**} = a^*$

Keyword = if|else|for|... **Comparison** = <|>|<=|>=|==|... **Whitespace** = $|\backslash t|\backslash n|\backslash t\backslash t|..\to$ $(\backslash t|\backslash n|\ )^*$ **Digit** = 0|1|...|9 **Integer** = Digit|DigitDigit|... -> $Digit^+$ **Letter** = a|b|...|z|A|B|...|Z

ID = (Letter|_)(Letter|Digit|_) * **Float** = $(\epsilon|-)Digit^+.Digit^+(\epsilon|E(\epsilon|+|-)Digit^+)$

1. 이런 token들 만들어 merge **Merged = Keyword | ID | Comp | Float | Whitespace | ...**

2. input stream $a_1a_2a_3 \ldots a_n$을 cursor 앞으로 옮겨가면서 L(Merged)에 속하는지 확인

Ex. $midx = 1,\ a_1 \in L(M)$ $midx = 2, a_1a_2 \in L(M) \ldots midx = 4, a_1a_2a_3a_4 \notin L(M)$

-> $a_1a_2a_3$ classify / a_4 partition 이 과정을 계속 반복

* classification에서 두 token에 속하면? -> priority / error handling **=> by Finite Automata**

Lecture 03. Lexical Analysis (Recognition of tokens)

Finite automata $M = \{Q, \Sigma, \delta, q_0, F\}$ | finite set of states $Q = \{q_0, q_1, \ldots, q_i\}$

Input alphabet Σ = finite set of input symbols | start state q_0

Set of accepting(final) states $F\ (\subset Q)$ | set of state transition functions δ

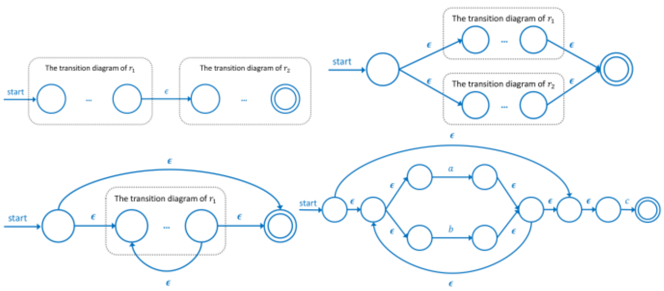
($\delta(q_0, a) = q_1$: state transition from q_0 to q_1 on input symbol a)

DFA는 ϵ-move 허용 X, 각 state에 각 input symbol마다 이동 O (최대 한 개)

NFA는 ϵ-move 허용 O, 각 state에 각 input symbol마다 여러 개의 이동 가능

	DFA	NFA
# of transitions per input per state	Zero or one	Zero or more
ϵ-move	X	O
# of path for a given input	Only one	One or more
Accepting condition	For a given input, its path must end in one of accepting states	For a given input, there must be at least one path ending in one of accepting states
Pros	Fast to execute (only one path) but complex	slow but Simple to represent (easy to make/understand)
Cons	Complex -> space problem (exponentially larger than NFA)	Slow -> performance problem (several paths)

Thomson's construction으로 표현해보자



NFA to DFA -> subset construction algorithm

각 Ter input symbol({a,b})에 대해 확인하기

$T_0 = \epsilon\text{-}closure(A) = \{A, B, C\}$

$T_1 = \epsilon\text{-}closure(\delta(T_0, a)) = \epsilon\text{-}closure(D) = \{D, F, G\}$

$T_2 = \epsilon\text{-}closure(\delta(T_0, b)) = \epsilon\text{-}closure(E) = \{E, F, G\}$

$T_3 = \epsilon\text{-}closure(\delta(T_1, a)) = \epsilon\text{-}closure(H) = \{H\}$ $(= \epsilon\text{-}closure(\delta(T_2, a)))$

$\epsilon\text{-}closure(\delta(T_1, b)) = \emptyset$

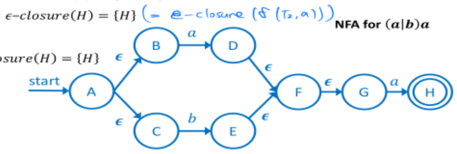
$\epsilon\text{-}closure(\delta(T_2, a)) = \epsilon\text{-}closure(H) = \{H\}$

$\epsilon\text{-}closure(\delta(T_2\ b)) = \emptyset$

$\epsilon\text{-}closure(\delta(T_3\ a)) = \emptyset$

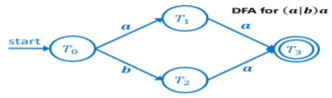
$\epsilon\text{-}closure(\delta(T_3\ b)) = \emptyset$

NFA for $(a|b)a$



	a	b
T_0	T_1	T_2
T_1	T_3	$\emptyset$
T_2	T_3	$\emptyset$
T_3	$\emptyset$	$\emptyset$

DFA for $(a|b)a$



Lecture 04. Syntax Analyzer (Parser) (Context Free Grammars)

CFG : Terminals, Non-terminals, start symbol, productions로 구성

Terminals: basic symbols (cannot be replaced)

Non-terminals: syntactic variables (can be replaced by other non-term or term)

Start symbol: one non-terminal | Productions: replacement rule

(대문자 alphabet -> non-terminal, 소문자 alphabet -> term)

(로마자 $\alpha, \beta \ldots$ -> sequence of non-term, term, ϵ ex. $\alpha = aABBBcddef$)

$(^n)^n$ => BALANCED -> (BALANCED) | ϵ good at recursive structure

Derivation (=>) : sequence of replacement (=>*: derivate zero or more times)

Rule: Leftmost ($=>_{\{lm\}}$) :replace left-most non-term first / Rightmost ($=>_{\{rm\}}$) : rightmost~

Token validation set-1) sentinel form of CFG G, 2) sentence of CFG G, 3) lang of CFG G

Definition: A sentinel form of a CFG G sequence of terminals

- α is a sentinel form of G, if $A \Rightarrow^* \alpha$, where A is the start symbol of G
  - If $A \Rightarrow^*_{lm} \alpha$ or $A \Rightarrow^*_{rm} \alpha$, α is a (left or right) sentinel form of G

Definition: A sentence of a CFG G

- α is a sentence form of G,

  if α is a sentinel form of a CFG G which consists of terminals only

Definition: A language of a CFG G

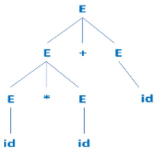
- $L(G)$ is a language of a CFG G (context-free language)
- $L(G) = \{\alpha \mid \alpha\ is\ a\ sentence\ of\ G\}$ set of sentence of cfg

If an input string (e.g., a token set) is in $L(G)$, we can say that it is valid in G

$E \rightarrow E + E \mid E * E \mid (E) \mid id$

For $id * id + id$

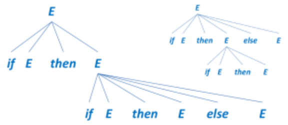
- E
- $\Rightarrow_{lm} E + E$
- $\Rightarrow_{lm} E * E + E$
- $\Rightarrow_{lm} id * E + E$
- $\Rightarrow_{lm} id * id + E$
- $\Rightarrow_{lm} id * id + id$



Good CFG? -> non-ambiguous / no left recursion / for each nonterminal, only one choice of production starting from a specific input symbol

Ambiguity -> cfg를 통해 한 string을 여러 개의 parse tree로 구성할 수 있을 때

$E \rightarrow if\ E\ then\ E \mid if\ E\ then\ E\ else\ E \mid other$



$E \rightarrow MATCHED \mid UNMATCHED$

$MATCHED \rightarrow if\ MATCHED\ then\ MATCHED\ else\ MATCHED \mid other$

$UNMATCHED \rightarrow if\ E\ then\ E \mid if\ E\ then\ MATCHED\ else\ UNMATCHED$

Left recursion - A =>+ Aα 인 경우 Infinite loop 돌게 됨 (sometimes)

Rewrite using **right-recursion** S->Sa|b를 S->bA, A->aA|ϵ 처럼 쓰는 것

$S \rightarrow S\alpha_1 \mid S\alpha_2 \mid \dots \mid S\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$ can be rewritten as:

Step 1: Make a new nonterminal A and add a production rule $\alpha_i A$ for all α_i and ϵ

- $A \rightarrow \alpha_1 A \mid \alpha_2 A \mid \dots \mid \alpha_m A \mid \epsilon$

Step 2: For a nonterminal S, add a production rule $\beta_i A$ for all β_i and discard other rules

- $S \rightarrow \beta_1 A \mid \beta_2 A \mid \dots \mid \beta_n A,\quad A \rightarrow \alpha_1 A \mid \alpha_2 A \mid \dots \mid \alpha_m A \mid \epsilon$

만약, same input symbol로부터 2개 이상의 productions 존재한다?

$E \rightarrow T + E \mid T,\ T \rightarrow F * T \mid F,\ F \rightarrow (E) \mid id$ -> Left Factoring으로 해결

$E \rightarrow T + E \mid T,\quad T \rightarrow F * T \mid F,\quad F \rightarrow (E) \mid id$

Step 1: For each non-terminal A, find the longest common prefix of productions α

- e.g., for E, $\alpha = T$

Step 2: Discard all productions which have the form of $A \rightarrow \alpha\beta$, and add $A \rightarrow \alpha A'$

- e.g., $E \rightarrow TE'$ $E \rightarrow \alpha E' \rightarrow TE'$ $T \rightarrow \alpha T' \rightarrow FT'$

Step 3: For the new non-terminal A', add $A' \rightarrow \beta$ for all discarded productions in step 2

- e.g., $E' \rightarrow +E \mid \epsilon$ $E' \rightarrow +E \mid \epsilon$ (∵ $E \rightarrow T + E \mid T$) $T' \rightarrow *T \mid \epsilon$

Step 4: Repeat step 1 ~ 3 until there is no more common prefix for all non-terminals

- $E \rightarrow TE',\quad E' \rightarrow +E \mid \epsilon,\quad T \rightarrow FT',\quad T' \rightarrow *T \mid \epsilon,\quad F \rightarrow (E) \mid id$

=> non-ambiguous, right recursive, left factoring 이 세 요소가 중요함!

G: DECL -> DECL type id; | DECL type id= id; | ϵ

Step 1: rewrite G by using right recursion

DECL → βA → A
A → α₁A | α₂A | ϵ

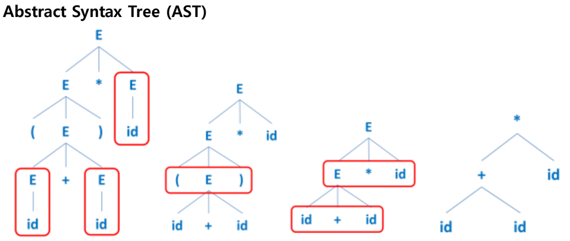
DECL → A
A → type id; A | type id = id; A | ϵ

Step 2: rewrite G by using left factoring

① α = type id
DECL → A
A → α; A | α = id; A | ϵ
A' → ;A | =id;A

② DECL → A
A → type id A' | ϵ
A' → ;A | =id;A

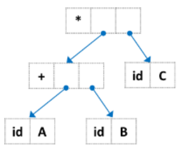
Abstract Syntax Tree (AST)



1. single-successor nodes 2. Symbols for describing syntactic details
3. Non-terminals with an operator and arguments as their child nodes

AST construction G: E -> E + E | E * E | (E) | id (id+id)*id

Production	Semantic action
E -> E1 + E2	E.node = new Node(' + ', E1.node, E2.node)
E -> E1 * E2	E.node = new Node(' * ', E1.node, E2.node)
E -> (E1)	E.node = E1.node
E -> id	E.node = new Leaf(id, id.value)

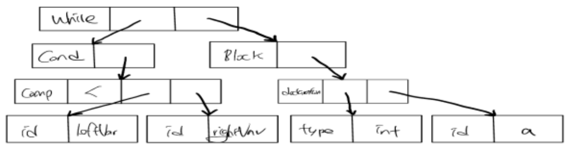


Example G:S -> while(C){B}, C->id comp id, B->type id; | id();

Production	Semantic action
$S \rightarrow while(C)\{B\}$	$S.node = new\ Node('while', C.node, B.node)$
$C \rightarrow id_1\ comp\ id_2$	$C.node = new\ Node('cond', new\ Node('comp', comp.value, new\ Leaf(id_1, id_1.value), new\ Leaf(id_2, id_2.value)))$
$B \rightarrow type\ id;$	$B.node = new\ Node('block', new\ Node('declaration', new\ Leaf(type, type.value), new\ Leaf(id, id.value)))$
$B \rightarrow id();$	$B.node = new\ Node('block', new\ Node('call', new\ Leaf(id, id.value)))$

while (leftVar < rightVar){int a;}

$S \Rightarrow_{lm} while(C)\{B\} \Rightarrow_{lm} while(id\ comp\ id)\{B\} \Rightarrow_{lm} while(id\ comp\ id)\{type\ id;\}$



Top-down Parsing (Leftmost derivation; LL parsing; Left-to-right / Leftmost)

#1 Recursive descent -> 순서대로 시도하다 문제 생기면 backtracking

-> non-ambiguous CFG, no left recursive CFG => possible but not effective (backtracking)

#2 LL(k) parsing 중 LL(1) -> non-ambiguous CFG, no left recursive CFG, **left factored CFG**

(A 로 시작하는 2 개의 규칙이 있으면, backtracking 또는 LL(2) 필요)

1) construct LL(1) parsing table 2) given input string 에 대해 parsing

LL(1) parsing table construction

First set of non-terminal A: $First(A) = \{x \mid A =>^* x\alpha\}$ A 의 derivation 중 처음 나올 수 있는 term 의 집합 $if\ A =>^* \epsilon,\ \ \epsilon \in First(A)$

First set of x (terminal) : First(x)={x} First set of α: First(α) i) First(α) = First(x) if $\alpha = x\beta$ ii) First(α) = First(A1) U First(A2) U … U First(An) U First(x) if $\alpha = A_1A_2 \dots A_n x\beta\ and\ \epsilon \in First(A_i) for\ all\ i$ iii) $\epsilon \in First(\alpha), if\ \alpha = A_1A_2 \dots A_n\ and\ \epsilon \in First(A_i)\ for\ all\ i$

$E \rightarrow TE',\quad E' \rightarrow +E \mid \epsilon,\quad T \rightarrow FT',\quad T' \rightarrow *T \mid \epsilon,\quad F \rightarrow (E) \mid id$

$First(F) = First((E)) \cup First(id) = First(() \cup First(id) = \{(, id\}$

$First(T') = First(*T) \cup First(\epsilon) = First(*) \cup First(\epsilon) = \{*, \epsilon\}$

$First(T) = First(FT') = First(F) = \{(, id\}$

$First(E') = First(+E) \cup First(\epsilon) = First(+) \cup First(\epsilon) = \{+, \epsilon\}$

$First(E) = First(TE') = First(T) = \{(, id\}$

Follow set of a non-terminal A: $Follow(A) = \{x \mid S =>^* \alpha Ax\beta\}$ derivation 중 A 의 바로 옆에 나올 수 있는 term 의 집합 $\$ \in Follow(S)$

$First(\beta) - \{\epsilon\} \subseteq Follow(A),\ \ if\ there\ is\ a\ production\ B \rightarrow \alpha A\beta$

$Follow(B) \subseteq Follow(A),\ \ if\ there\ is\ a\ production\ B \rightarrow \alpha A\beta, where\ \epsilon \in First(\beta)$
$or, if\ there\ is\ a\ production\ B \rightarrow \alpha A$

$E \rightarrow TE',\quad E' \rightarrow +E \mid \epsilon,\quad T \rightarrow FT',\quad T' \rightarrow *T \mid \epsilon,\quad F \rightarrow (E) \mid id$

$First(F) = \{(, id\},\quad First(T') = \{*, \epsilon\},\quad First(T) = \{(, id\},$

$First(E') = \{+, \epsilon\},\quad First(E) = \{(, id\}$

F → (E) | id E' → +E | ϵ

- $Follow(E) = \{\$\} \cup First()) \cup Follow(E') = \{\$,)\} \cup Follow(E) = \{\$,)\}$
- $Follow(E') = Follow(E) = \{\$,)\}$ Follow(E) ⊆ Follow(E') & Follow(E') ⊆ Follow(E)
- $Follow(T) = First(E') - \{\epsilon\} \cup Follow(E) \cup Follow(T') = \{+, \$,)\} \cup Follow(T) = \{+, \$,)\}$
- $Follow(T') = Follow(T) = \{+, \$,)\}$
- $Follow(F) = First(T') - \{\epsilon\} \cup Follow(T) = \{*, +, \$,)\}$

For each terminal $x \in First(\alpha)$, Fill the table entry $[A, x]$ as α

For each terminal $x \in Follow(A)$, Fill the table entry $[A, x]$ as α, if $\epsilon \in First(\alpha)$

		The next input symbol					
		+	*	(	)	id	$ (endmarker)
Leftmost non-terminal	E			TE'		TE'	
	E'	$+E$					
	T			FT'		FT'	
	T'		$*T$				
	F			(E)		id	

		The next input symbol					
		+	*	(	)	id	$ (endmarker)
Leftmost non-terminal	E			TE'		TE'	
	E'	$+E$			ϵ		ϵ
	T			FT'		FT'	
	T'	ϵ	$*T$		ϵ		ϵ
	F			(E)		id	

Bottom-up Parsing

LR parsing L: Left-to-Right scan of input R: rightmost derivation

1) Left factoring X 2) Left-recursive elimination X 3) unambiguous O

$bool\ BUParsingWithBacktracking(string\ \alpha)\{$

$SStr = \{\beta \mid \beta\ is\ a\ substring\ of\ \alpha\ and\ \beta\ can\ be\ reduced\ by\ a\ non-terminal\}$

$(there\ is\ a\ production\ X \rightarrow \beta_i)$

$for\ each\ \beta_i \in SStr$

$replace\ \beta_i\ by\ its\ corresponding\ non-terminal\ and\ store\ the\ result\ as\ \alpha'$

$if\ (\alpha' == S) || (BUParsingWithBacktracking(\alpha') == true),\quad return\ true;$

end recursive

$return\ false;$

$\}$

$if\ BUParsingWithBacktracking(inputString) == true,\ accept$

$otherwise,\ \ reject$

$S \rightarrow dAc \mid cAe \mid cAd,\quad A \rightarrow a$

Check $BUParsingWithBacktracking(cad)$

- $SStr = \{a\}$ (there is a production $A \rightarrow a$)
- $\alpha' = cAd$ (replace a in cad by A)
- Check $BUParsingWithBacktracking(cAd)$
  - $SStr = \{cAd\}$ (there is a production $S \rightarrow cAd$)
  - $\alpha' = S$ (replace cAd in cAd by S)
  - **Accept!!**